

AD-A249 417



**RL-TR-91-274, Vol I (of five)
Final Technical Report
November 1991**



2

PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Developing Formally Verified Ada Programs

ORA Corporation



**Sponsored by
Strategic Defense Initiative Office**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.

92-11269



**Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700**

92 4 27 412

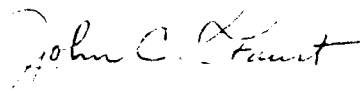
This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Although this report references limited documents listed below, no limited information has been extracted:

RL-TR-91-274, Vol IIIa, IIIb, IVa, and IVb, November 1991. Distribution authorized to USGO agencies and their contractors; critical technology; Nov 91.

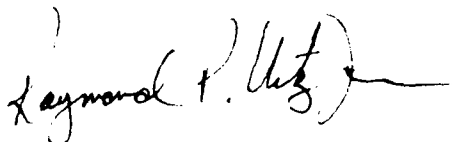
RL-TR-91-274, Vol I (of five) has been reviewed and is approved for publication.

APPROVED:



JOHN C. FAUST
Project Engineer

FOR THE COMMANDER:

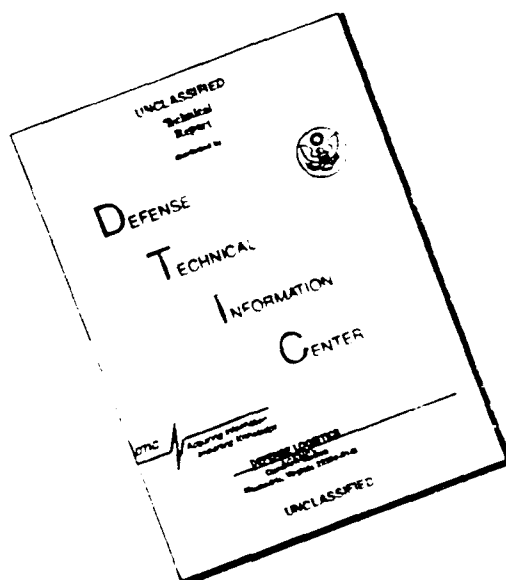


RAYMOND P. URTZ, JR.
Director
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3AB) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST
QUALITY AVAILABLE. THE COPY
FURNISHED TO DTIC CONTAINED
A SIGNIFICANT NUMBER OF
PAGES WHICH DO NOT
REPRODUCE LEGIBLY.

PENELOPE: AN ADA VERIFICATION ENVIRONMENT,
Developing Formally Verified Ada Programs

Norman Ramsey

Contractor:	ORA Corporation
Contract Number:	F30602-86-C-0071
Effective Date of Contract:	19 Aug 86
Contract Expiration Date:	30 Sep 89
Short Title of Work:	PENELOPE: AN ADA VERI- FICATION ENVIRONMENT, Developing Formally Veri- fied Ada Programs
Period of Work Covered:	Aug 86 - Aug 89
Principal Investigator:	Maureen Stillman
Phone:	(607) 277-2020
RL Project Engineer:	John C. Faust
Phone:	(315) 330-3241

Approved for public release; distribution unlimited.

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by John C. Faust, RL/C3AB, Griffiss AFB 13441-5700, under contract F30602-86-C-0071.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE November 1991		3. REPORT TYPE AND DATES COVERED Final Aug 86 - Aug 89	
4. TITLE AND SUBTITLE PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Developing Formally Verified Ada Programs				5. FUNDING NUMBERS C - F30602-86-C-0071 PE - 35167G/63223C PR - 1070/B413 TA - 01/03 WU - 02	
6. AUTHOR(S) Norman Ramsey					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ORA Corporation 301A Dates Drive Ithaca NY 14850-1313				8. PERFORMING ORGANIZATION REPORT NUMBER ORA TR 17-3	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Office, Office of the Secretary of Defense Wash DC 20301-7100 Rome Laboratory (C3AB) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-91-274, Vol I (of five)	
11. SUPPLEMENTARY NOTES RL Project Engineer: John C. Faust/C3AB/(315) 330-3241					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) <p>Odyssey Research Associates has undertaken a study of the feasibility of developing formally verified Ada programs. We have designed a specification language for sequential Ada programs. It is a member of the Larch family of specification languages. We have built a prototype program editor that is intended to help programmers develop programs and proofs from specifications, as advocated by Dijkstra and Gries (2,4). It contains predicate transformers, which compute wp (an approximation to the weakest precondition of a program), and it generates verification conditions.</p> <p>The semantics of the specification language and the definition of the predicate transformers are derivable from a denotational definition of sequential Ada. The predicate transformers can be proved sound with respect to these definitions by structural induction on programs. The denotational-style definition of the predicate transformers is well suited to an implementation as an attribute grammar.</p> <p>The program editor is designed to be used on program fragments, not just complete programs. The next step in improving the prototype editor is to find ways to simplify the intermediate values of wp so they can be used to guide the development of fragments into programs.</p>					
14. SUBJECT TERMS Ada, Larch, Larch/Ada, Formal Methods, Formal Specification, Program Verification, Predicate Transformers, Ada Verification				15. NUMBER OF PAGES 48	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
				20. LIMITATION OF ABSTRACT UL	

Developing Formally Verified Ada Programs*

Norman Ramsey[†]
Odyssey Research Associates

October 10, 1988

Abstract

Odyssey Research Associates has undertaken a study of the feasibility of developing formally verified Ada programs. We have designed a specification language for sequential Ada programs. It is a member of the Larch family of specification languages. We have built a prototype program editor that is intended to help programmers develop programs and proofs from specifications, as advocated by Dijkstra and Gries [2,4]. It contains predicate transformers, which compute wp (an approximation to the weakest precondition of a program), and it generates verification conditions.

The semantics of the specification language and the definition of the predicate transformers are derivable from a denotational definition of sequential Ada. The predicate transformers can be proved sound with respect to these definitions by structural induction on programs. The denotational-style definition of the predicate transformers is well suited to an implementation as an attribute grammar.

The program editor is designed to be used on program fragments, not just complete programs. The next step in improving the prototype editor is to find ways to simplify the intermediate values of wp so they can be used to guide the development of fragments into programs.

Introduction

Writing formal specifications of programs and proving that programs meet those specifications should help programmers develop more reliable software.

*This research has been sponsored by the USAF, Rome Air Development Center, under contract number F30602-86-C-0071.

[†]Current address: Department of Computer Science, Princeton University, Princeton, New Jersey 08544

Edsger Dijkstra and David Gries have stressed that a program should not face verification as a hurdle after development, but should be developed in such a way as to ensure its correctness [2,4]. One should begin with a formal specification, and the development of the program and its proof should follow from that specification. Gries, in his *Science of Programming*, suggests ways to use a specification to guide the development of a program and its proof.

We have undertaken to build software that will help programmers apply the methods of formal development advocated by Gries and Dijkstra. The software is designed to lead to formal verification tools with three properties:

- The tools should not just help to verify finished programs or to check proofs of such programs, but should help programmers to develop verified programs.
- The tools should be based on sound, explicitly stated mathematics.
- The tools should support programming in a subset of Ada, since there is a need for reliable software written in Ada.

Overview of results

We have designed a specification language, Larch/Ada-88, for sequential Ada programs. We have implemented a prototype of an editor, Penelope, which will help programmers develop and verify programs specified with Larch/Ada-88.¹ The prototype implementation supports a subset of Ada that is roughly "PASCAL with exceptions." We have completed some of the mathematics that supports the Larch/Ada-88 definition and the Penelope implementation.

Specification language The Larch/Ada-88 specification language is part of the Larch family of two-tiered specification languages [5,19].² The two-tiered approach separates the specification of individual program modules from the specification of underlying abstractions. The Larch Shared

¹The specification language was formerly known as PolyAnna. We have changed the name of the language to Larch/Ada because it is a Larch interface language. We have also given the name Penelope to our prototype verification system.

²We use "specification" in the traditional sense of a statement of requirements. What the Ada Language Reference calls "specifications" should be thought of as like "declarations" in other programming languages.

Language is used to specify the underlying abstractions: for example, it can be used to define the notions of array, list, set, bag, and so on. A Larch Shared Language specification defines a set of terms (and some theorems about the terms); this set becomes the *assertion language* used in Larch/Ada-88 specifications.

Larch/Ada-88 (henceforth Larch/Ada) is a Larch *interface language*; it is used to specify Ada programs by attaching assertions at certain points, like the entry and exit points of subprograms. The specification constructs of Larch/Ada are called *annotations*, since most of them are derived from similar constructs in Anna [10].

Implementation Most verification tools work in batch mode [7,3]. The user writes a program, supplies a specification and appropriate assertions, and then submits it all to a *verification condition generator*. The resulting verification conditions must be shown to hold. A verification condition for a program is an assertion whose truth guarantees that the program satisfies its specification.

When a programmer using a batch system makes a refinement or discovers a mistake, the whole job must be resubmitted, and correct work may have to be redone. Moreover, most batch systems cannot verify program fragments. As practiced by Gries and Dijkstra, program development consists largely in building up correct fragments by accretion, and in refining existing fragments. Batch verification fits poorly with these techniques.

We have replaced the traditional batch verification system with a program editor, which we call Penelope. Using Penelope, programmers can examine weakest preconditions as they are computed and can use them to guide program development. Program fragments can be proved correct, then picked up and placed in larger contexts.

We have built the Penelope editor by using the Cornell Synthesizer Generator [17,16]. Penelope can be used to create and edit abstract syntax trees that represent parts of an annotated Ada program. The interface is that of a traditional syntax-directed editor in which the user may use a mouse or an EMACS-like command set to manipulate trees. Penelope computes weakest preconditions as attributes of the nodes of the syntax tree.³ Attribute evaluation is incremental: that is, every time the user changes the tree, weakest preconditions are recomputed where necessary, and the new

³Actually, as in Gries [4], the system works with an *approximation* to the weakest precondition, called *wp*.

By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

preconditions are available for display. When certain Ada constructs are used, Penelope generates a verification condition; typically one verification condition is generated for each subprogram and one for each loop.

A user of Penelope begins by writing down a formal specification of an Ada subprogram using Larch/Ada. (The formal specification takes the form of an Ada "specification" augmented with *subprogram annotations*.) He or she then builds the subprogram body, working backwards as described by Gries [4]. Penelope can be instructed to display weakest preconditions at any point, and also to display any verification conditions that may be generated. The user can alter the program and immediately observe the effects on preconditions and verification conditions, since the recomputation of *wp* and of verification conditions is incremental and automatic. (The preconditions and verification conditions are simplified somewhat before being presented to the user.)

The verification conditions are sentences in pure logic; their statement is independent of program context. If they can be shown to hold, the program containing them satisfies its specification. Penelope contains a proving component that can be used to prove facts about integers, Booleans, and Ada types like arrays and records. The proving component is primitive; the proof of a program like binary search takes up many pages.

Formal foundations The formal foundation of Larch/Ada has two parts, one dealing with assertions and another dealing with annotations. The assertion part covers the semantics of the Larch/Ada assertion language as defined using Larch Shared Language specifications. This semantics is determined entirely by the semantics of the Larch Shared Language, and is therefore independent of Ada. The assertion part also covers the formal specification (using the Larch Shared Language) of Ada's data types.

The annotation part connects the execution semantics of sequential Ada with the annotations of Ada programs defined by Larch/Ada. The Penelope editor's weakest precondition computations are based on a formal statement of *predicate transformers*. The predicate transformers define a function from a Larch/Ada specification and an annotated Ada program to a set of verification conditions. The transformers are based on a continuation semantics for the sequential part of Ada. As part of this work, Polak [14] has shown how to establish a formal connection between a continuation semantics and predicate transformers. To define the predicate transformers, he lets the

denotation of a program be a function on terms of the Larch/Ada assertion language.

Larch/Ada

Two-tiered specifications In the two-tiered system of specification, the *shared language* component is used to define all the abstractions used in specification and verification. For example, in an arbitrary-precision arithmetic package, the shared language part of the specification would define what we mean by integer and by addition. One might implement arbitrary-precision arithmetic using the idea of registers of arbitrary length. The shared language would be used to define what we mean by a register and its length.

The *interface language* part of a specification is used to state what a program does in terms of the abstractions defined in the shared language part. The interface language part of the arbitrary-precision arithmetic package would show which subprograms perform what operations on the data, and would state that overflow never occurs.

Using the two-tiered system, a designer can keep the unpleasant details introduced by the programming language isolated in the interface language component, while doing the real intellectual work of specification in the shared language component. It is possible to use the shared language to say exactly what is meant by, for example, a stack or a directed graph, without getting bogged down in details of exceptional conditions or of representation.

The shared language part of a specification defines the assertion language used in the interface language part of the specification. Formulas in this assertion language are formulas in first-order predicate logic. When we attempt to prove that an implementation satisfies its specification, we first apply the predicate transformers, which are based on the definition of Ada, to a program and its specification. The predicate transformers produce a verification condition, which is a sentence in the assertion language. The proof of the verification condition needs to refer only to the shared language part of the specification; no further reference to the definition of Ada is required. Since the assertion language is really a particular formulation of first-order logic, checking the correctness of the proofs of verification conditions is straightforward.

Languages in the two-tiered system Three languages are important in the two-tiered system of specification. The Larch Shared Language enables users to write formal specifications of useful abstractions. It is a single language, used to write specifications. The Larch Shared Language part of a specification consists of one or more *traits*, each one of which may specify several abstractions. The shared language part of a specification defines an *assertion language*, which is the language used to refer to the abstractions. (A sentence stating that a stack is not empty would be a Boolean term in the assertion language.) In general, every shared language specification defines a different assertion language, although the different assertion languages have much in common, since they are just different formulations of first-order logic.

Larch/Ada is the language in which we write the *interface language* part of a specification. This part uses *annotations* to specify the behavior of a program. The annotations contain assertions, which are formulas in the assertion language defined by the shared language part of the same specification.

Description of Larch/Ada The assertion language defined by a particular shared-language specification is essentially a particular predicate calculus. We will describe the features of the Larch/Ada interface language, then give an example.

The simplest Larch/Ada annotation is the *embedded assertion*. The assertion is a formula in the current assertion language (defined by the shared language part of the current specification). Free variables in the assertion refer to program variables. Embedded assertions may be inserted at certain control points in an implementation (i.e. between statements or between declarations); they constrain the implementation to satisfy the assertion whenever control reaches that point. In other words, the embedded assertion says that, whenever control reaches it, if we substitute the actual values of program variables into the assertion, it must denote truth. The embedded assertion can be used only in an implementation, as a guide to the predicate transformers; it cannot be used in specifications.

Larch/Ada's *subprogram annotations* are used both to constrain subprogram implementations, and to specify subprograms. The entry/exit kinds are the *in annotation*, the *out annotation*, and the *result annotation*. Each of these contains a single assertion. They are used to constrain states on

entry to a subprogram, to constrain states on exit from a subprogram, and to constrain the values a function might return.

Also among the subprogram annotations are several *exception propagation annotations*, which are used to say when exceptions may and may not be raised, when exceptions must be raised, and what conditions must hold when exceptions are raised. Finally there is the *side effect annotation*, which is used to document side effects on or dependence on global variables.

Larch/Ada has two features which help make the annotation mechanism more powerful. The first is the "IN variable." Within a subprogram, parameter names can be modified by IN to denote the value of the parameter on entry to the subprogram. IN variables are essential in subprogram specifications, because we almost always need to refer to the initial values of parameters. For example, here are specifications for some of the familiar stack operations:

```
PROCEDURE push(s: IN OUT stack; x: element);
--| WHERE
--|     OUT s = push(IN s,x);
--| END WHERE;
```

```
PROCEDURE pop(s: IN OUT stack);
--| WHERE
--|     IN NOT is_empty(s);
--|     OUT s = pop(IN s);
--| END WHERE;
```

The other enriching feature of Larch/Ada is a way to introduce "virtual variables."⁴ These variables are not used in writing specifications, but are defined within implementations to help in proofs of correctness. Their values don't actually affect the results of a computation. Larch/Ada allows the user to declare virtual variables, to assign to them, and to use them in annotations.

Finally, in support of data abstraction (packages with private types), Larch/Ada enables the user to define *abstraction functions* using the Larch Shared Language, and to associate these abstraction functions with Ada types, using the *based on* annotation. Thus, a user writing an arbitrary-precision arithmetic package might implement registers using arrays, and

⁴These are called "ghost variables" by Gries and Dijkstra; the name "virtual variable" is from Luckham [10].

Figure 1: Shared language specification of set operations (Wing [19])

```

SetOfE: trait
  includes Integer
  introduces
    empty:  $\rightarrow$  SI
    add: SI, E  $\rightarrow$  SI
    remove: SI, E  $\rightarrow$  SI
    has: SI, E  $\rightarrow$  Bool
    isEmpty: SI  $\rightarrow$  Bool
    card: SI  $\rightarrow$  Int
  constrains empty, add, remove, has, isEmpty, card so that
  SI generated by [empty, add]
    for all [s: SI, e, e1: E]
      remove(empty, e) = empty
      remove(add(s, e), e1) =
        if e = e1 then remove(s, e1) else add(remove(s, e1), e)
      has(empty, e) = false
      has(add(s, e), e1) = if e = e1 then true else has(s, e1)
      isEmpty(empty) = true
      isEmpty(add(s, e)) = false
      card(empty) = 0
      card(add(s, e)) = if has(s, e) then card(s) else 1 + card(s)

```

would then define an abstraction function from arrays to registers. The proofs of correctness of implementations of abstract data types are as described by Hoare [8]; the abstraction function is used to rewrite a specification that was in terms of an abstract type to a new specification in terms of a concrete type.

Using Larch/Ada As an example, we will present a Larch/Ada specification for some set operations. Jeannette Wing used these operators in her presentation of a Larch interface language for CLU [19]. As she did, we will use a Larch Shared Language description of sets and set operations. The trait describing these operations, called **SetOfE**, is shown in Figure 1.

We begin our example with the specification of a “choose” procedure that selects a member of a set, removes it from the set, and returns it. Since Ada functions may not have side effects on their parameters, we will formulate the returned member as an OUT parameter:

```
--| WITH SetOfE WITH [set FOR si, integer FOR e];
PROCEDURE choose (IN OUT s: set; i: OUT integer);
--| WHERE
--|     IN NOT IsEmpty(s);
--|     OUT has(IN s, i) AND s = remove(IN s, i);
--| END WHERE;
```

The `--| WITH` annotation specifies that `SetOfE` is the trait that defines the notion of set and the operations `IsEmpty`, `has`, and `remove`. The `IN` annotation states that `choose` may be called only on nonempty sets. The `OUT` annotation gives the relation between the initial and final values of the set `s` and the final value of the integer `i`. (Notice that the final values are specified by giving just the variable name, while the initial values are specified by modifying the name by `IN`. In the `IN` annotation, all variables are implicitly modified by `IN`.) Because no side effect annotation is present, `choose` may not modify or read any global variables.

Figure 2 shows a Larch/Ada specification for a set package. This package specifies the same operators as the similar example in Wing [19].

Formal foundation of Larch/Ada and Penelope

Connecting Ada to a denotational model Formal verification of Ada programs must be based on a formal definition of the Ada language itself, but at this time there is no official formal definition of Ada. We circumvent this difficulty by providing a denotational model of a computing language `Ada'`, and by considering Larch/Ada to be a specification language for `Ada'`. Ada and `Ada'` have the same syntax, and we argue informally that for a restricted class of programs and computations they have the same observable behavior.

We restrict Ada most by omitting from `Ada'` all features involving concurrency. While there is widespread consensus on what are good methods to model and specify sequential imperative languages, there is no similar consensus on the utility of the various proposed methods of modeling and specifying concurrent programs. We have omitted other features from `Ada'` because they are machine-dependent (e.g. representation clauses) or because

Figure 2: Larch/Ada specification for a set package

```
--| WITH SetOfE WITH [integer FOR e];
PACKAGE sets IS
  TYPE set IS PRIVATE; --| based on si;

  FUNCTION pair(i, j : integer) RETURN set;
    --| WHERE
    --|      RETURN add(add(empty, i), j);
    --| END WHERE;

  PROCEDURE union(s1 : set; s2 : IN OUT set);
    --| WHERE
    --|      OUT (FORALL j::((has(s2, j)=has(IN s1, j))
    --|                                     OR has(IN s2, j)));
    --| END WHERE;

  PROCEDURE intersect(s1 : set; s2 : IN OUT set);
    --| WHERE
    --|      OUT (FORALL j::((has(s2, j)=has(IN s1, j))
    --|                                     AND has(IN s2, j)));
    --| END WHERE;

  FUNCTION member(s : set; i : integer) RETURN boolean;
    --| WHERE
    --|      RETURN has(s, i);
    --| END WHERE;

  FUNCTION size(s : set) RETURN integer;
    --| WHERE
    --|      RETURN card(s);
    --| END WHERE;
END sets;
```

it is not feasible to formalize them (e.g. the exact circumstances under which `storage_error` is raised). Here is a partial list of omitted features:

- Concurrency
- Real number types
- Representation specifications and other implementation-dependent or machine-dependent features
- Unchecked conversion and unchecked deallocation
- The predefined exceptions `storage_error` and `numeric_error` (i.e. we consider that no execution of an Ada' program ever raises these exceptions), and computations that result in "undetected numeric overflow"
- Optimizations that cause execution of Ada statements in other than the canonical order
- Parameter aliasing in procedure calls
- Any program called *erroneous* by the Ada reference manual

Some of these restrictions (e.g. that forbidding aliasing) can be enforced by suitable static checks.

Although Ada' is not Ada, they are intended to be equivalent within our area of interest, and we will not distinguish them in what follows.

Connecting the denotational model to Larch/Ada and the predicate transformers The predicate transformers implemented in Penelope are derived from a continuation semantics for Ada. The task of defining a continuation semantics for Ada has been considerably simplified by two expedients. First, the static semantics of Ada is not part of the definition; the definition assumes a suitably checked and attributed abstract syntax representation of programs. Second, the semantics of the Ada types is not part of the definition; instead, the semantics of Ada types is defined by Larch Shared Language specifications. The technique used for deriving predicate transformers from a denotational semantics is one developed by Polak [13,14]. The current definition of the predicate transformers used in Penelope does not provide for proofs of termination.

Here we give an example that shows what we mean when we say that Larch/Ada and Penelope are formally based. The example suggests how we

define the semantics of Larch/Ada and how we show that the VC generation implemented in Penelope is sound.

Generating verification conditions involves manipulating a number of different languages. Since a VC generator takes as input a program and a specification, and produces as output an assertion (the verification condition), the fundamental languages are the programming language P , the specification language S , and the assertion language A . For simplicity, we take the assertion language as fixed, although it is actually determined using the shared language part of the specification.

The actual input to Penelope is a specification together with an *annotated* implementation; we will call that conglomerate V , and we define projection functions that extract the relevant parts:

$$\begin{array}{ll} \pi : V \rightarrow P & \text{Extract the implementation \textit{without} annotations} \\ \sigma : V \rightarrow S & \text{Extract the specification} \end{array}$$

Then the VC generator is a function $vcgen : V \rightarrow A$, and we wish to show that $vcgen$ is sound, i.e. whenever the verification condition $vcgen(v)$ holds, the program πv satisfies its specification σv .

We need to consider the denotations of the various syntactic objects we have been discussing. For simplicity, we'll let the denotation of a program be a mapping from states to states (this makes sense even for a continuation semantics if one considers whole programs). We'll call B the special Boolean domain consisting of the two elements truth and falsehood (which we'll write as t and f). If the set of states is X , we have

$$\begin{array}{ll} M_P : P \rightarrow (X \rightarrow X) & \text{A program denotes a state changer.} \\ M_S : S \rightarrow ((X \rightarrow X) \rightarrow B) & \text{A specification denotes a predicate on state changers.} \\ M_A : A \rightarrow (X \rightarrow B) & \text{An assertion denotes a predicate on states.} \end{array}$$

Intuitively, a program is a function from states to states, a specification defines a predicate ("satisfaction") on programs, and an assertion defines a predicate on states (also called "satisfaction"). A program $p \in P$ satisfies a specification $s \in S$ if

$$(M_S s)(M_P p) = t.$$

The semantics of the specification language can be defined in terms of the semantics of the programming and assertion languages, M_P and M_A , provided we have a fixed (not necessarily finite) set of states X . If the specification language is a simple one that gives only entry and exit assertions,

i.e. $S = A \times A$, then we can define

$$M_S(a_1, a_2)\rho = \forall x \in X. M_a a_1 x \Rightarrow M_a a_2(\rho x),$$

where $\rho = M_P(p)$ is a state transformer. The definition says that a program satisfies its specification if, whenever the entry condition a_1 holds on entry to the program, the exit condition a_2 holds on exit from the program. (We are using \Rightarrow to denote mathematical implication.)

The semantics of Larch/Ada is defined in two steps: first, we define a mapping from Larch/Ada to a simpler language in which a specification consists of an entry condition, an exit condition for normal termination, and exit conditions for termination by raising exceptions. In the second step we define the semantics of the simpler language; this language is essentially $S = A \times \text{list } A$, and the definition of its semantics is very similar to that just shown.

As the discussion above suggests, V is not really fundamental; we introduced V to stand for the input to *vcgen*. If we have an (annotated program, specification) pair $v \in V$, we are really only interested in the projections πv and σv . The program satisfies its specification when

$$M_S(\sigma v)(M_P(\pi v)) = \text{true}.$$

VC generation is sound if the truth of the verification condition guarantees that the program satisfies its specification. The truth of a verification condition must be independent of state, so the function *vcgen* is sound if, for any $v \in V$,

$$\forall x \in X (M_A(\text{vcgen}(v))) \Rightarrow M_S(\sigma v)(M_P(\pi v)).$$

Polak [14] goes into more detail, giving a complete definition of *vcgen* for a small programming language P . He describes a way of deriving *vcgen* from the semantics of P and sketches a proof of soundness that uses structural induction on programs. The techniques he describes are the same ones used to define the semantics of Larch/Ada and to prove the soundness of the predicate transformers implemented in Penelope.

The Penelope implementation

We are implementing tools that support research in formal verification using Larch/Ada. Of these, the most important is the Penelope editor, which

helps users develop verified programs. In the prototype, there is no support for writing traits: every specification has the same shared language part, and there is a fixed assertion language used for all specifications. The prototype can be used to specify and prove Ada subprograms, provided those subprograms use no global variables. We plan to extend the Penelope prototype to read definitions of extensions to the assertion language. We plan to supplement it with a tool that will help prove Penelope verification conditions.

Status of the implementation

Penelope is implemented using the Cornell synthesizer generator [17]. The Cornell synthesizer generator accepts as input a description of an attribute grammar and compiles this description into a syntax-directed editor which can compute and display the values of attributes. The heart of this editor is an algorithm which, when the edited tree is changed, computes and propagates the changes in attribute values [16].

We think of Penelope as having three components: predicate transformation, proving, and simplification. Predicate transformation is central. That component reads and interprets the Larch/Ada annotations, computes wp , and generates verification conditions. The user controls which intermediate values of wp are displayed and which verification conditions are displayed. The displayed values are updated every time the user changes his or her program or specification.

The proving component is a sub-editor that enables the user to construct proofs of the verification conditions, using a sequent calculus. The editor presents a list of hypotheses and a goal, and the user designates an inference rule to apply. The application may generate subgoals, and the process continues until the subgoals are reduced to axioms, which are automatically recognized by the editor. The editor has built in a small number of proof tactics; the user can designate one of these tactics instead of designating a rule.

The simplification component is a set of functions that can be called by the other two components. These functions make the preconditions and verification conditions more readable. One function does this by rewriting; terms like $P \wedge \text{true}$ are rewritten to P , and so on. Another function attempts to find ways to substitute simpler terms for terms that are especially complex or hard to read. Another performs arithmetic operations on integer literals. Several functions manipulate the forms of terms in order to make other operations easy; depending on circumstances, one may prefer $P \supset (Q \supset$

$(R \supset S))$ to $P \wedge Q \wedge R \supset S$, or vice versa. (We are using \supset to represent the implication symbol in the assertion language.) Collectively, these functions reduce the size of the verification conditions Penelope generates.

The currently supported subset of Ada We have imposed a number of restrictions on programs editable with Penelope, as described earlier. Concurrency, real number types, parameter aliasing, `storage_error`, and some other features are forbidden. Here are the highlights of what Penelope does with those programs it accepts:

- Subprograms may call predefined or user-defined subprograms; recursion is supported.
- Arbitrary user-defined exceptions, `raise` statements, and exception handlers are supported.
- Integer, Boolean, enumeration, array, and record types are supported. The predefined Ada operators for these types are supported. (Subtyping is not supported.)
- All of the Ada control structures are supported except `goto` statements, `case` statements, and `for` loops.
- Some static semantic checking is performed, including type checking and overload resolution.

The current Penelope is limited in what it can prove. Many capabilities which might be considered difficult have been investigated mathematically but have not yet been implemented. These include:

- Proving that neither of the predefined exceptions `constraint_error` and `program_error` is ever raised
- Proving that programs terminate
- Proving that programs are not erroneous
- Detecting potential aliasing and illegal order dependencies by suitable static semantic checking
- Proving programs that define subtypes whose bounds are set dynamically
- Proving programs that may raise or handle the predefined exceptions `constraint_error` and `program_error`

The currently supported subset of Larch/Ada The current implementation of the Penelope editor supports most of Larch/Ada. The major features that are missing are those associated with termination, global variables and side effects, and data abstraction. As noted earlier, there is no support for writing traits in the Larch Shared Language. This means that the shared language part of every specification is the same, and that there is a single assertion language used for all Larch/Ada specifications. That assertion language is restricted to terms describing integers, Booleans, arrays, and records. (Enumeration literals are converted to integers during predicate transformation.)

Future plans

Simplification The greatest weakness of the current Penelope editor is that weakest preconditions and verification conditions are too hard to read. The simplification component is good at reducing the bulk and complexity of Boolean terms; most of the complexity in verification conditions comes from arithmetic terms. Simplifying such terms is the next step in improving our implementation. Rather than build an arithmetic simplifier from scratch, we plan to connect an existing simplifier to the Penelope editor. The simplifier that we plan to use is based on the Nelson-Oppen procedure for combining decision procedures [12,15].

Extending Penelope In order to support data abstraction, we intend to add packages and private types to Penelope's Ada subset. (Adding these constructs is straightforward; the major difficulties involved in supporting data abstraction arise in extending the assertion language, as discussed below.) We may also add new control structures, in particular the **case** statement and the **for** loop.

In the longer term, we plan to add support for proofs of termination and for proofs of subprograms that have side effects on global variables.

Data Abstraction It is not possible to write readable formal specifications of large programs without taking advantage of data abstraction. Generating verification conditions for programs that use abstract data types is not hard, but generating sound verification conditions for implementations of abstract data types can be tricky. This is especially true in Ada, where the abstraction constructs do not completely hide the representation.

It is also hard to simplify and prove verification conditions when specifications refer to abstract data types. The essence of the difficulty is that, when using data abstraction, the user must add new terms to the assertion language. (These terms describe the new abstractions, like stacks, buffers, registers, or whatever may be needed to specify a particular application. They are introduced and defined by traits written in the Larch Shared Language.) To be able to make effective use of the new terms, we must be able to show that their introduction does not lead to any logical inconsistency. We must also be able to extend the proving and simplification components of Penelope to be able to handle the new terms.

We will begin studying data abstraction by making our assertion language extensible. We will use a tiny subset of the Larch Shared Language, a subset which will enable us to add to the assertion language new sort and operator symbols. In particular, it will be possible to add abstract sorts and abstraction functions to the assertion language. We will then allow users to make assertions (without proof) involving the new symbols they have introduced. In proofs of programs, these assertions will be treated like axioms. We hope that, by studying the kinds of assertions users make, we will be able to learn what methods of proof might help users prove programs that use data abstraction.

Conclusions

Our efforts have been concentrated on defining Larch/Ada-88 and on building the Penelope prototype. Evaluation of Larch/Ada and Penelope must await the completion of the prototype and experience with its use, but we can draw some conclusions about the methods we have applied and about the difficulty of the problems that remain.

We have developed a useful technique for deriving predicate transformers, and we have developed a method for implementing the transformers using an attribute grammar. We have some preliminary observations about the results of attempting to mechanize Gries's and Dijkstra's methods of program development. Finally, we believe we have learned what problems need to be solved before a useful verification system can be built.

Implementing predicate transformers The denotational style of writing predicate transformers lends itself to a natural and efficient implementation of the transformers as an attribute grammar. Values in the transformer

definition map to attributes of the grammar, and meaning functions map to the semantic equations that define the relationships among the attributes. We can avoid implementing lambda-abstraction and beta-reduction for the language of terms by using pairs of attributes to represent values of arrow types.

Mechanizing formal development The biggest obstacle to learning Gries's method of program development is the drudgery of computing wp . This difficulty increases as the complexity of the programming language increases; it would be unrealistic to expect to compute wp by hand for a language like Ada. Fortunately, it is easy to mechanize the computation of wp , provided a denotational-style definition of wp is available.

The problem with a mechanized computation of wp is that the resulting preconditions quickly become too complicated to be understood by a human being, at which point they can no longer be used to guide program development, which is the whole point of Gries's method. We have yet to learn whether mechanical simplifiers like the one described by Nelson and Oppen can make the preconditions understandably simple. The problem doesn't arise when wp is computed by hand, because in that case the programmer constantly applies his or her knowledge of integers, sequences, Booleans, and so forth, so that computation and simplification proceed simultaneously.

Open problems The Larch/Ada specifications we can write using Penelope are limited by the fixed, non-extensible assertion language. (It is a severe limit; for example, at this time we cannot introduce the factorial function for use in a specification.) The programs we can prove using Penelope are limited by the size of the weakest preconditions Penelope computes. We believe that the most important problem remaining to be solved is the one of being able to introduce new terms into an assertion language, while simultaneously introducing methods of simplification and proof for those terms. The Larch Shared Language provides a way of writing formal definitions of new terms. We need to develop a formal representation of methods of proof and simplification. Finally, we need to develop ways of showing that the addition of new definitions introduces no logical inconsistency, and ways of showing that the proof and simplification methods are consistent with the definitions.

Related work

AFFIRM, built at USC-ISI, was the first verification system to use algebraic specifications and a rewrite rule prover [11]. The Gypsy system was the first verification system to handle a form of concurrency [7]. The Stanford Pascal Verifier was the first verification system to handle a real programming language [9]. The most important contribution of the Stanford Pascal Verifier project was probably the Nelson-Oppen method of combining decision procedures [12].

The Anna project is an effort to introduce formal specification to Ada programmers by providing specification constructs which can be checked at run time [10]. The aim of the AVA project is to define a verifiable subset of Ada and to give it a formal semantics using Boyer-Moore logic [18.1].

Acknowledgements

The work described herein was done at Odyssey Research Associates with Wolfgang Polak, Carla Marceau, David Guaspari, C. Douglas Harper, and Doug Weber. Anna provided large repository of specification constructs, on which we drew heavily when designing Larch/Ada. The Anna group at Stanford was forthcoming with suggestions about how formal verification of Ada programs might proceed. John Guttag helped us to understand Larch and to explore how we might adapt Larch to Ada. In particular, he helped us elucidate the issues that needed to be addressed in defining the semantics of a Larch interface language. John Guttag and Steve Garland lent us their theorem prover, *lp*, which we used in our study of verification conditions generated by Penelope.

David Guaspari and Wolfgang Polak helped me understand the formal foundations of Larch/Ada and Penelope, and they corrected many errors in earlier drafts of this paper.

References

- [1] R. S. Boyer and J. S. Moore. "Proving Theorems about LISP Functions," *JACM* 22, 1, 129-144.
- [2] Edsger W. Dijkstra. *The Discipline of Programming*. Prentice-Hall, 1976.

- [3] J. Crow, S. Jefferson, R. Lee, M. Melliar-Smith, J. Rushby, R. Schwartz, R. Shostak, and F. von Henke, *Preliminary Definition of the revised SPECIAL specification language*, SRI International, 1986.
- [4] David Gries, *The Science of Programming*, Springer-Verlag, 1981.
- [5] J. V. Guttag, J. J. Horning, and J. M. Wing, *Larch in Five Easy Pieces*, DEC/SRC TR 5, July 1985.
- [6] S. Garland, J. Guttag, and J. Staunstrup, "Verification of VLSI Circuits using LP", manuscript.
- [7] D. I. Good, R. L. Akers, and L. M. Smith, *Report on Gypsy 2.05*, Computational Logic Inc., 1986.
- [8] C. A. R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica 1*, pp 271-281 (1972).
- [9] D. C. Luckham *et al.*, *Stanford Pascal Verifier User Manual*, Report No. STAN-CS-79-731, Stanford University, March 1979.
- [10] D. C. Luckham *et al.*, *Anna: A Language for Annotating Ada Programs*, Reference Manual, 1986.
- [11] D.R. Musser, "Abstract Data Type Specifications in the AFFIRM System," in *Proceedings of the Specifications of Reliable Software Conference*, IEEE Computer Society (April 1979), 47-57.
- [12] G. Nelson and D. C. Oppen, "Simplification by Cooperating Decision Procedures", *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct. 1979), 245-257.
- [13] Wolfgang Polak, "Program Verification Based on Denotational Semantics," POPL '81.
- [14] Wolfgang Polak, "A Technique for Writing Predicate Transformers," submitted to LICS '89.
- [15] T. Redmond, *Simplifier Description*, Aerospace Technical Report ATR-85 (8354)-8, Nov. 1985.
- [16] Thomas Reps, *Generating Language-Based Environments*, MIT Press, 1984.

- [17] Thomas Reps and Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, 1987.
- [18] Michael K. Smith, "A Verifiable Ada," Formal Methods Committee Report, *Ada Letters* 8, 4 (July/August 1988), 136-142.
- [19] J. M. Wing, "Writing Larch Interface Language Specifications," *ACM Trans. Program. Lang. Syst.* 9, 1 (Jan. 1987), 1-24.

DISTRIBUTION LIST

addresses	number of copies
RL/C3WA ATTN: John C. Faust Griffiss AFB NY 13441-5700	25
ORA CORPORATION 301A Dates Drive Ithaca NY 14850-1515	5
RL/DCVL Technical Library Griffiss AFB NY 13441-5700	1
Administrator Defense Technical Info Center DTIC-FOAD Cameron Station Building 5 Alexandria VA 22304-6145	2
Strategic Defense Initiative Office Office of the Secretary of Defense Wash DC 20301-7100	2
RL/C2AF Griffiss AFB NY 13441-5700	1
HQ USAF/SCIT Washington DC 20330-5100	1
SAF/ASCC Pentagon 2m 4000 AF3 Wash DC 20300	1

Naval Warfare Assessment Center 1
GIDEP Operations Center/Code 316
ATTN: E Richards
Corona CA 91720

HQ AFSC/XTH 1
Andrews AFB MD 20334-5020

HQ SAC/SCPT 2
OFFUTT AFB NE 68146

HQ TAC/DRIY 1
ATTN: Maj. Divine
Langley AFB VA 23065-5575

HQ TAC/DDA 1
Langley AFB VA 23065-5554

ASD/THMS 1
Wright-Patterson AFB OH 45433-6503

SM-ALC/MACEA 1
ATTN: Danny McClure
Bldg 237, MAGEF
McClellan AFB CA 95552

WRDC/AAAI-4 1
Wright-Patterson AFB OH 45433-6543

WRDC/AAAI-2 1
ATTN: Mr Franklin Hutson
WPAFB OH 45433-6543

AFIT/LDEF
Building 642, Area B
Wright-Patterson AFB OH 45433-6593

1

WRDC/MTCL
Wright-Patterson AFB OH 45433

1

AAWRL/HE
Wright-Patterson AFB OH 45433-4573

1

Air Force Human Resources Lab
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45437

1

AUL/LSE
Bldg 1405
Maxwell AFB AL 36112-5564

1

HQ ATC/ITDI
ATTN: Lt Col Killian
Randolph AFB TX 78150-1001

1

AFLMC/LGV
ATTN: Maj. Chaffer
Building 205
Gunter AFB AL 36114-6000

1

US Army Strategic Def
CSSD-14-PA
PO Box 1500
Huntsville AL 35897-1500

1

Ofc of the Chief of Naval Operation ATTN: William J. Cook Navy Electromagnetic Spectrum Dir Room 54574, Pentagon (52-241) Wash DC 20350	1
Commanding Officer Naval Avionics Center Library 0/765 Indianapolis IN 46219-0100	1
Commanding Officer Naval Ocean Systems Center Technical Library Code 95424 San Diego CA 92152-5000	1
Cmde Naval Weapons Center Technical Library/03431 China Lake CA 93555-4001	1
Superintendent Code 524 Naval Postgraduate School Monterey CA 93943-5000	1
Space & Naval Warfare Systems Comm Washington DC 20363-5100	1
CDR, U.S. Army Missile Command Redstone Scientific Info Center AMSMI-PD-CS-2/ILL Documents Redstone Arsenal AL 35893-5241	1
Advisory Group on Electron Devices 231 Varick Street, Rm 1140 New York NY 10014	2
Los Alamos National Laboratory Report Library MS 5000 Los Alamos NM 87544	1

AEDC Library 1
Tech Files/MS-100
Arnold AFB TX 37339

Commander, USAR 1
ASDH-PCA-CRL/Tech Lib
Bldg 61801
Ft Huachuca AZ 85613-6000

1839 EIG/EIT 1
Keesler AFB MS 39574-6348

AFEWG/ESRI 3
San Antonio TX 78243-5000

ESD/YRR 1
Hanscom AFB MA 01731-5000

SAI JPR 1
ATTN: Major Charles J. Ryan
Carnegie Mellon University
Pittsburgh PA 15213-1690

Director NSA/CSS 1
TS122/TOL
ATTN: D W Mirjanum
Fort Meade MD 21755-6000

Director NSA/CSS 1
W157
9800 Savage Road
Fort Meade MD 21755-6000

NSA 1
ATTN: D. Alley
Div X911
9800 Savage Road
Ft Meade MD 20755-6000

Director 1
NSA/CSS
W11 DEFSMAC
ATTN: Mr. Mark E. Clesh
Fort George G. Meade MD 20755-6000

Director 1
NSA/CSS P12
ATTN: Mr. Dennis Heinbuch
9800 Savage Road
Fort George G. Meade MD 20755-6000

DDO 1
P31
9800 Savage Road
Ft. Meade MD 20755-6000

DIPNSA 1
P509
9800 Savage Road
Ft Meade MD 20755

Director 1
NSA/CSS
P33/P 3 3 ELDS
Fort George G. Meade MD 20755-6000

DDO Computer Center 1
C/TIC
9800 Savage Road
Fort George G. Meade MD 20755-6000

ESD/AV 1
HANS COM AFR MA 01731-5000

ESD/IC 1
HANS COM AFR MA 01731-5000

FL 2PC7/RESEARCH LIBRARY
DL A4/SULL
HANSCOM AFB MA 01731-5000

1

TECHNICAL REPORTS CENTER
MAIL DROP D133
BURLINGTON ROAD
BEDFORD MA 01731

1

SDI/S-P1-AM
ATTN: Cmdr Korajo
The Pentagon
Wash DC 20311-7100

1

SDIO/S-PL-PM
ATTN: Capt Johnson
The Pentagon
Wash DC 20301-7000

1

SDIO/S-PL-RM
ATTN: Lt Col Rindt
The Pentagon
Wash DC 20301-7100

1

SDI Technical Information Center
1755 Jefferson Davis Highway #705
Arlington VA 22202

1

SAF/AWSD
ATTN: Maj M. K. Jones
The Pentagon
Wash DC 20331

1

AFSC/CV-D
ATTN: Lt Col Flynn
Andrews AFB MD 20374-5000

1

HQ SD/X2
ATTN: Col Heinich
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

HQ SSO/CNC
ATTN: Col O'Brien
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

HQ SSO/CNCI
ATTN: Col Collins
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

SSO/AT
ATTN: Col Ryan
Hanscom AFB MA 01731-5000

1

SSO/ATS
ATTN: Lt Col Oldenber;
Hanscom AFB MA 01731-5000

1

SSO/ATN
ATTN: Col Leib
Hanscom AFB MA 01731-5000

1

AFSTC/XPX (Lt Col Detucci)
Kirtland AFB NM 87117

1

AFSPACCOM/XPB
ATTN: Maj Roger Hunter
Peterson AFB CO 80914

1

SSD/CNI
ATTN: Lt Col Joe Rouge
P. O. Box 92960
Los Angeles AFB CA 90009-2960

1

NTR JPO
ATTN: Maj Don Savenscroft
Falcon AFB CO 80912

1

Naval Air Development Ctr 1
ATTN: Dr. Mort Metersky
Code 300
Warminster PA 18974

HQ AFOTEC/OAHS 1
ATTN: Dr. Samuel Charlton
Kirtland AFB NM 87117

ESD/XTS 1
ATTN: Lt Col Joseph Toole
Hanscom AFB MA 01731

SDIC/ENA 1
ATTN: Col R. Worrell
Pentagon
Wash DC 20301

USA-SDC CDSO-H-33E 1
ATTN: Mr. Doyle Thomas
Huntsville AL 35897

HQ AFSPACECOM/DOXP 1
ATTN: Capt Jack Terrace
Stop 7
Peterson AFB CO 80914

ESD/XTE 1
ATTN: Lt Col Paul Monico
Hanscom AFB MA 01731

CSSD-H-53 1
ATTN: Mr. Larry Tubbs
Commander USA SDC
PO Box 1500
Huntsville AL 35897

USSPACECOM/JSP 1
ATTN: Lt Col Harold Stanley
Peterson AFB CO 80914

VTR JPC 1
ATTN: Mr. Nat Sojourner
Falcon AFB CO 2L712

PAOC/OJA 1
ATTN: Mr. Anthony F. Snyder
Griffiss AFB NY 13441

AF Space Command/KPXIS 1
Peterson AFB CO 80914-5001

AFOTEC/XPP 1
ATTN: Capt Wrobel
Kirtland AFB NM 87117

Director NSA (V31) 1
ATTN: George Hoover
9800 Savage Road
Ft George G. Meade MD 20755-6000

SSD/CNIR 1
ATTN: Capt Brandenburg
PO BOX 92263-2960
LOS ANGELES CA 90009-2960

National Computer Security Center 1
ATTN: C4/TIC
9800 Savage Road
Fort George G Meade MD 20755-6000

Unisys Corp/Network Info Sys Div 1
ATTN: Lorraine Martin
5151 Camino Ruiz
Camarillo CA 93610

Mitre Corp 1
ATTN: Dale M. Johnson (MS A047)
Burlington Rd
Bedford MA 01730-0208

Secure Computing Technology Corp 1
ATTN: J. Thomas Haigh
1210 West County Road E. (Ste 107)
Arden Hills MN 55112

Mitre Corp 1
ATTN: Joshua Guttman (MS 4040)
Burlington Rd
Bedford MA 01730-0208

Mitre Corp 1
ATTN: John Janeri (MS 4047)
Burlington Rd
Bedford MA 01730-0208

DRA Corporation 1
ATTN: Dr. Richard Platek
301A Harris B. Dates Dr.
Ithaca NY 14850-1313

National Security Agency 1
ATTN: Larry Hatch/R5
9300 Savage Rd
Ft Meade MD 20755-6000

National Security Agency 1
ATTN: Mark Woodcock/R233
9300 Savage Rd
Ft Meade MD 20755-6000

Naval Research Laboratory 1
ATTN: Carl F. Landwehr (Code 5542)
Wash DC 20375-5000

US Army Communication-Electronics 1
Command
ATTN: AMSEL-20-CR-IS-P
ATTN: John W. Preusse
Ft Monmouth NJ 07703

NASA Langley Research Center 1
ATTN: Ricky Butler (MS 133)
Hampton VA 23665-0225

National Security Agency 1
ATTN: Michele Pittelli/R233
9800 Savage Rd
Ft Meade MD 20755-6000

National Security Agency 1
ATTN: Howard Stainer/R23
9800 Savage Rd
Ft Meade MD 20755-6000

SRAWAR/Code 7242 1
ATTN: Bob Kolacki
Wash DC 20363-5100

Naval Research Laboratory 1
ATTN: John McLean (Code 5540)
Wash DC 20375-5000

ASD/YFAF (Ms Swengim) 1
Wright-Patterson AFB OH 45433-6503

Naval Research Laboratory 1
Code 55403
ATTN: H. G. Lubes
Wash DC 20375-5000

DARPA/ISTO 1
ATTN: Dr. William Scherlis
1400 Wilson Blvd
Arlington VA 22209-2304

DARPA/ISTO 1
ATTN: Dr. Jack Kramer
1400 Wilson Blvd
Arlington VA 22209-2306

SEI JPO 1
ATTN: Maj Charles J. Ryan
Carnegie Mellon University
Pittsburgh PA 15213-3890

Defense Communications Agency 1
ATTN: Dr. Cass Defiore (Code ACA)
Advanced Technology Office
Wash DC 20305-2000

SDIO/SDA 1
ATTN: Lt Col Jim Sweeder
Pentagon, Room 1E149
Wash DC 20310

ESD/AVSE 1
ATTN: Capt Roland Lesieur
Hanscom AFB MA 01731-5000

Trusted Information Systems, Inc. 1
ATTN: Richard E. Schwainger
P.O. Box 45
3060 Washington Rd
Glenwood MD 21738

SRI International 1
ATTN: Darlene Sherwood
For: Comp Sci Lab/John Rushby
333 Ravenswood Ave
Menlo Park CA 94025

Institute for Defense Analysis 1
ATTN: Dr. Russell Fries
Comp & Soft Eng Div/4. Mayfield
1501 N. Beauregard Street
Alexandria VA 22311-1770

Secure Computing Technology Corp 1
ATTN: Jerry A. Herby
1210 W. County Rd E, Suite 100
Arden Hills MN 55112

SRI International 1
ATTN: Darlene Sherwood
For: Comp Sci Lab/Teresa Lunt
333 Ravenswood Ave
Menlo Park CA 94025

The Aerospace Corp/Def Develop Div 1
ATTN: James G. Gee
For: George Gilley, WL-146
P.O. Box 22957
Los Angeles CA 90009-2957

GE Co/Strategic Systems Dept
ATTN: Tim Pawlik
For: Bill Samsch
1787 Sentry Park Way PO Box 1000
Bluebell PA 19422

1

GE Co/Strategic Systems Dept
ATTN: Tim Pawlik
For: Mr. Don Marking
1787 Sentry Park Way PO Box 1000
Bluebell PA 19422

1

Advisory Group on Electron Devices
201 Varick Street, Rm 1140
New York NY 10014

1

MITRE Corp
ATTN: Dr. Donna Cuomo
Bedford MA 01730

1

Ford Aerospace Corp
c/o Rockwell International
ATTN: Dr. John Schulz
1250 Academy Park Loop
Colorado Springs CO 80910

1

Essex Corp
ATTN: Dr. Bob Mackie
Human Factors Research Div
5775 Dawson Ave
Goleta CA 93117

1

RJO Enterprises
ATTN: Mr. Dave Israel
1225 Jefferson Davis Hwy
Suite 707
Arlington VA 22202

1

BBN Systems & Technology
ATTN: Dr. Dick Paw
70 Fawcett St
Cambridge MA 02138

1

Bonnie McDaniel, MDE
313 Franklin St
Huntsville AL 35891

1

Harris Corp
Government Info Sys Division
ATTN: Ronda Henning
PO Box 98000
Melbourne FL 32902

1

Ford Aerospace & Comm Corp
ATTN: Peter Wake (Mail Stop 294)
10440 State Highway 53
Colorado Springs CO 80918

1

Computational Logic, Inc.
ATTN: Dr. Donald L. Good
1717 W. 5th St (Suite 200)
Austin TX 78701

1

Gemini Computers Inc.
ATTN: Roger Schnell
2511 Garden Rd (Bldg C, Ste 1001)
Monterey CA 93941

1

Boeing Aerospace Co
ATTN: Dan Schnackenberg (MS 3F-10)
P.O. Box 3999
Seattle WA 98124-2499

1

RSA Laboratories, Inc.
ATTN: Steve Vinter
10 Moulton Street
Cambridge MA 02138

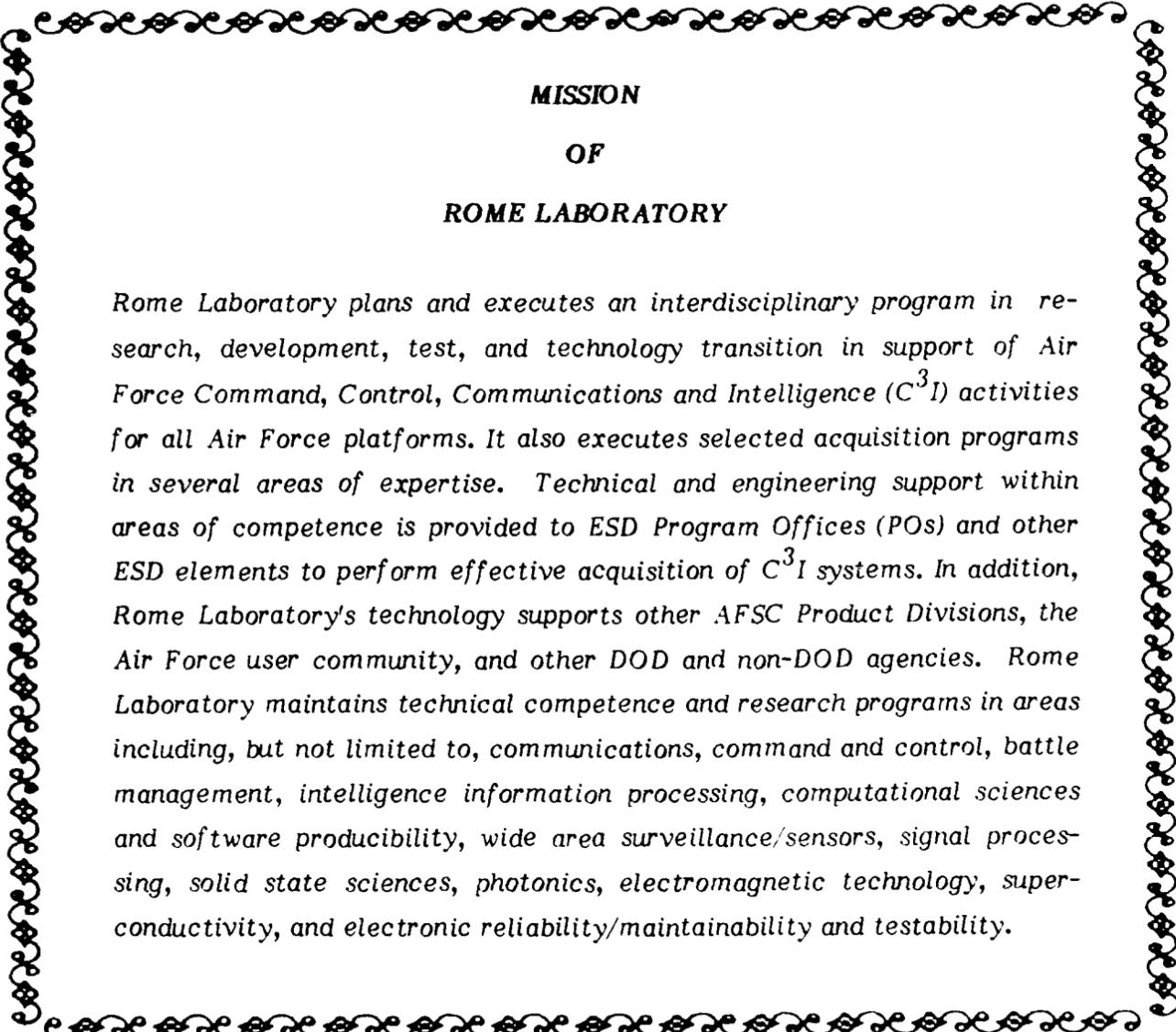
1

Univ of Calif at Santa Barbara
Computer Science Dept
ATTN: Prof Richard W. Hammerer
Santa Barbara CA 93106

1

Unisys Corp
ATTN: Deborah Cooper
5731 Glendon Ave
Culver City CA 90230

1



**MISSION
OF
ROME LABORATORY**

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C³I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.